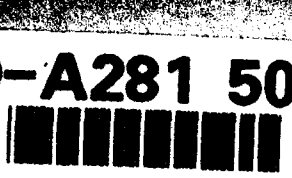


Classification in Feature-Based Default Inheritance Hierarchies

Marc Light

Technical Report 473
November 1993

UNIVERSITY OF
ROCHESTER
COMPUTER SCIENCE

Classification in Feature-based Default Inheritance Hierarchies

Marc Light

The University of Rochester
Computer Science Department
Rochester, New York 14627

Technical Report 473

November 1993

Abstract

When one works with a system that utilizes inheritance hierarchies the following problem often arises. A new object is introduced and it must be integrated into a hierarchy; under which classes in the hierarchy should the new object be positioned? In this paper, I formalize this problem for feature-based default inheritance hierarchies. Since it turns out to be NP-complete, I present an approximation algorithm for it. I show that this algorithm is efficient and look at some of the possible problematic situations for the algorithm. Although more analysis and experimentation are needed, these preliminary results show that the algorithm warrants such efforts.

This work was funded by NSF grant IRI-9013160, ONR/DARPA research grant no. N00014-92-J-1512, and a grant from the Land Baden-Württemberg, Germany.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	November 1993	Technical Report

4. TITLE AND SUBTITLE

Classification in Feature-based Default Inheritance Hierarchies

5. FUNDING NUMBERS

N00014-92-J-1512

6. AUTHOR(S)

Marc Light

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

The University of Rochester
Computer Science Dept.
734 Computer Studies Bldg.
Rochester, NY. 14627-0226

8. PERFORMING ORGANIZATION REPORT NUMBER

TR473

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Office of Naval Research
Information Systems
Arlington, VA 22217

DARPA
3701 N Fairfax Dr.
Arlington, VA. 22203

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Distribution of this document is unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

See title page.

14. SUBJECT TERMS

classification; inheritance; defaults; approximation algorithm;

15. NUMBER OF PAGES

29

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
unclassified	unclassified	unclassified	UL

1 Introduction

Recent computational linguistics research into natural language lexicons has gone beyond simply listing idiosyncratic information about words. Many researchers are adding some sort of structure to their lexicons to replace the old practice of simply listing, in an entry for a word, all its properties. This structure often takes the form of feature-based default inheritance hierarchies. For example, such hierarchies are being utilized in lexicons for speech processing [Gibbon and Bleiching, 1991], natural language understanding systems [Andry *et al.*, 1992; Copestake *et al.*, 1991; Russell *et al.*, 1992; Krieger and Nerbonne, 1991], lexicography [Kilgarriff, 1993], and theoretical linguistics [Flickinger and Nerbonne, 1992; Gibbon, 1990; Reinhard, 1990; Cahill, 1993].

The move to structured lexicons was motivated, in part, by the fact that many current linguistic theories (*e.g.*, Head Phrase Structure Grammar [Pollard and Sag, 1987]) relegate much of the complexity of natural language to the lexicon. Thus, linguistic generalizations about phonology, morphology, and, increasingly, syntax and semantics are being made in the lexicon. Another reason for the move to structured lexicons is that the use of machine readable dictionaries and corpora-based linguistics has made it possible to construct large lexicons relatively easily (*e.g.*, [Copestake *et al.*, 1991]). Large lexicons make issues like ease of maintenance and efficient use of memory more important. Inheritance hierarchies are one way to address these issues.

Many of the linguistic generalizations that need to be made in the lexicon involve a cluster of properties and often a hierarchy of these clusters. For example, the group of verbs often called transitive verbs share the following properties: they require a subject and an object, the subject must be a noun phrase, the subject must have nominative case, the object is usually a noun phrase, and the object usually has accusative case. Feature-based default inheritance hierarchies use features to represent the properties of a word and classes to group features. Features are most often some variant of attribute-value pairs (*e.g.*, [subj/case,accusative][1] meaning that the subject's case is accusative). Classes may stand in an inheritance relationship: the lower class (subclass) receives the features of the higher class (superclass). In a default hierarchy, the inheritance of any particular feature may be blocked by a feature local to the subclass. In addition, many systems allow a class to have multiple superclasses.

An example of a lexical hierarchy is presented in Figure 1. It encodes generalizations about complementation and control features of verbs in its classes and inheritance links. INCOMPLETE and EQUI are classes in the hierarchy. The pairs written directly below these classes are some of the features that can be inherited from them. Features inherited by a class can in turn be inherited from it (*i.e.*, inheritance is recursive). The lines between classes represent inheritance links. For example, all the features of INCOMPLETE are inherited by EQUI—unless overridden by features in CONTROL or in EQUI.

When using such structured lexicons, a problem arises in deciding to which classes a word belongs. This is an instance of the more general problem of placing a new class in the most appropriate place in a classification scheme. In order to avoid overloading the term 'classification', I will call this 'the insertion problem.' An example of the insertion problem is the task of placing the ditransitive form of *give* into the hierarchy of Figure 1.

For example, suppose that this form of *give* (*e.g.*, ***she gave him the ball***) has the following features.

[1] The first element of this pair is an atomic symbol. Here and throughout this report, the '/' has only mnemonic relevance.

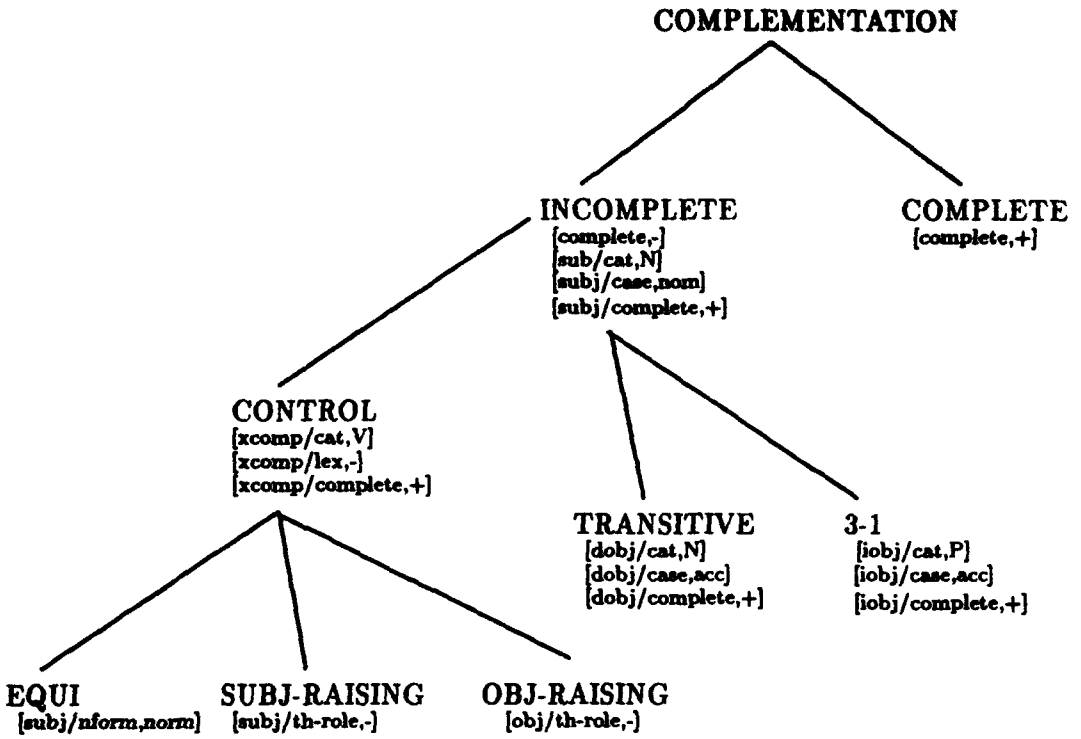


Figure 1: Complementation and control features of verbs (adapted from [Flickinger, 1987])

(1) [subj/cat,N]
[subj/case,nom]
[subj/complete,+]
[dobj/cat,N]
[dobj/case,acc]
[dobj/complete,+]
[iobj/cat,N]
[iobj/case,acc]
[iobj/complete,+]

A solution to this instance of the insertion problem would be to place *give* under the TRANSITIVE and 3-1 classes. By doing so, all of the features listed above could be inherited except [iobj/cat,N]. This feature would have to be listed directly in the entry for *give* so that the incorrect default inheritance of [iobj/cat,P] from the 3-1 class would be blocked. Thus, only three pieces of information need to be listed in the entry for the ditransitive verb *give*: it inherits from TRANSITIVE, it inherits from 3-1, and it has the feature [iobj/cat,N]. Notice that if we had inserted the entry under any other set of classes, more information would have had to be listed in the entry. The main characteristic of a good insertion is that it minimizes the amount of information that needs to be listed in the entry.

In lexicon research, the need for an automated insertion system arises in at least two situations. First, when designing a structured lexicon, it is necessary to test the generalizations the structure represents. One can do this by inserting large numbers of words and noting how well the words fall into the classes of the structure. If there are a large number of words that have similar features but do not fall into a single class then, perhaps, a class should be created for them. Or if a class is seldom used by the words for which it was created, then perhaps it should be modified in some way to better fit these words (this situation often arises when two or more classes overlap in the

generalizations they represent). Second, when using a finished hierarchy or a set of hierarchies to build a large lexicon, one must transfer large numbers of lexical items from raw data files or some other classification scheme to the hierarchies. For example, in the ELWIS project [Feldweg and Storrer, 1992] a relational database is used to store application-independent information which has been extracted from corpora and a machine readable dictionary. If the target application uses a feature-based default hierarchy, then an insertion system is needed to move the data from the relational database to the hierarchy. Figure 2 illustrates this situation. The first input, marked by the dotted line, to a general insertion system is the specific hierarchy into which the items will be inserted. After this hierarchy has been processed, the items themselves are taken from the database and inserted into the inheritance hierarchy.

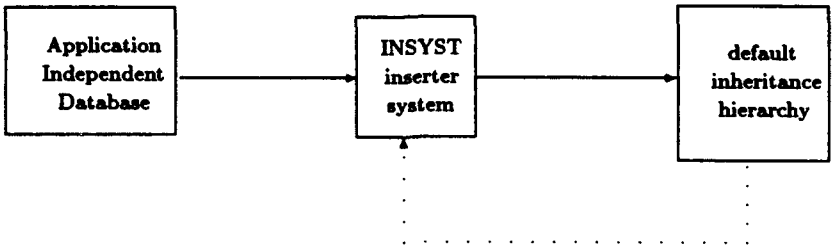


Figure 2: Transferring objects from a database to an inheritance hierarchy

In the following sections of this report, I will describe my approach to defining and solving the insertion problem for feature-based default inheritance hierarchies. First, I give an informal characterization of the problem and argue that this characterization captures the relevant aspects of the problem. Second, I formalize this characterization and in appendix A show that the problem is NP-complete. This result means that it is unlikely that a computationally tractable algorithm for its solution will be found. Next, I describe an approximation algorithm for the problem and through complexity analysis and discussion of preliminary experimentation, I argue that this algorithm produces reasonable results in an acceptable amount of time and space. This algorithm was modified slightly so that it would produce insertions for hierarchies encoded in the DATR formalism. This modified algorithm is part of a prototype of the INSYST system [Light *et al.*, 1992] which inserts lexical items taken from the ELWIS relational database [Feldweg and Storrer, 1992] into feature-based default hierarchies built in the DATR formalism. Appendix B contains example runs of the INSYST system.

Although this report focuses on the use of default inheritance hierarchies in natural language lexicon research, the results described here are relevant to any system that uses default inheritance hierarchies and requires automated insertion.

2 Informal Characterization of the Problem

A good insertion places an object under classes from which it can inherit most of its features. In addition, it should use as few classes as possible to achieve this goal. These two requirements are aspects of a basic principle of classification schemes: reduce redundancy. A corollary of this principle is that a good insertion should minimize the amount of information stored in the entry for the object being inserted and thereby maximize the use of the information contained in the inheritance relationships. Three types of information are stored in an entry: its superclass(es), the features that are not inherited from any superclass, and the features that are needed to block an incorrect inheritance. Thus, an optimal set of superclasses for the object being inserted has the smallest possible value for the following sum.

(2) number of superclasses + number of object features that are not in these superclasses + the number of object features that must be listed to block incorrect inheritance

I will refer to this sum as the cost of the insertion or solution.

At first glance, one might think that one easy way to find an optimal solution is to start at the roots of the hierarchies that make up a structured lexicon and simply walk down these hierarchies, pruning off the branches below a class that contains a feature that the object does not have (since all classes below this class will also have this feature). This approach will work for *strict* inheritance hierarchies. Strict hierarchies do not allow default inheritance—all the subclasses below a given class have to inherit all its features. The algorithm outlined above uses this characteristic to cut down the search space for superclasses. However, default hierarchies do not have this characteristic: if a class A inherits from a class B, the set of features associated with A might not be a superset of the set associated with B. Thus, an insertion algorithm for default hierarchies cannot ignore A as a possible superclass for an object simply because it has decided B is unsuitable; A might be an exception to B in just the right ways (see Figure 3). Because of these considerations, for the purposes of insertion,

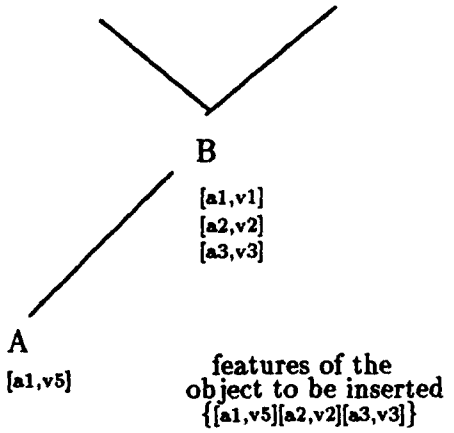


Figure 3: Insertion in default hierarchies

I make the following claim.

(3) Each class in a default hierarchy should be viewed as the set of features that can be inherited from this class.

(3) is a central claim of the approach to insertion taken in this report. (After I explain exactly what is meant by (3), I will defend the claim.) It amounts to compiling out the inheritance relationships so that a hierarchy becomes a set of sets of features.[2] 'Compiling out' the inheritance relationships is a process of pushing features down the inheritance links to the classes below so that all the features of a class are *explicitly* listed in the data structure for the class. For example, compiling out the hierarchy in Figure 1 would produce the set of sets listed in (4). The first set corresponds to the COMPLEMENTATION class, the second to INCOMPLETE, the third to COMPLETE, the fourth to TRANSITIVE, and the fifth to 3-1.

[2] Additional information such as the relative height from which a feature is inherited can be computed during this compilation process and attached to a feature via a weight. These weights could then be used when computing the cost of a solution. Such weights would be one way to add back a small amount of hierarchical information without making (3) vacuous.

(4) {{}
{[complete,-]
[subj/cat,N]
[subj/case,nom]
[subj/complete,+]}
{[complete,+]}
{[complete,-]
[subj/cat,N]
[subj/case,nom]
[subj/complete,+]
[dobj/cat,N]
[dobj/case,acc]
[dobj/complete,+]}
{[complete,-]
[subj/cat,N]
[subj/case,nom]
[subj/complete,+]
[iobj/cat,N]
[iobj/case,acc]
[iobj/complete,+]}
....
}

Now that we have a better idea of what is meant by (3), I will argue that (3) should be adopted when dealing with the problem of insertion. Intuitively, it might seem that reducing a hierarchy to a flat set of sets defeats the purpose of the hierarchy. This intuition exists because most hierarchies are built so that lower classes represent sub-generalizations of the generalizations represented by higher classes. These relations between generalizations are used by the hierarchy to reduce redundancy. By flattening the hierarchy, one loses these relationships and thus redundancy increases, which is usually bad. This intuition is correct when one considers storage and maintenance of information. More succinctly, if one uses a set of sets instead of a hierarchy to represent information, the ability to represent higher order generalizations is lost.

However, when one is concerned with insertion, the loss of higher order generalizations is irrelevant. It is irrelevant because if one accepts that a good insertion minimizes the space needed to store the properties of the object being inserted, then, with respect to insertion, a class is simply a chance to save space by storing one class name instead of a number of features. Thus, what features can be inherited from a class is the only characteristic of a class relevant to insertion. In order to compute which features can be inherited from any given class, one has to look at the entry for the class and its superclasses and in turn, their superclasses etc. At each level, the features of the classes above are passed down to the classes below. This is the processes of compiling out the hierarchy.

In response to this line of argumentation, one might counter that the hierarchy can be used to reduce the search space of possible superclasses. As mentioned above, this is true for strict hierarchies. However, as illustrated in Figure 3, this is *not* the case with default hierarchies. As a result, the inheritance links between classes in a default hierarchy lose their usefulness after it has been determined which features can be inherited from each class. Thus, when dealing with the insertion problem in default hierarchies, one should think of the hierarchy as a set of sets, each of which corresponds to a possible superclass of the object being inserted. The insertion problem, then, amounts to picking an optimal subset of this set of sets; the definition of optimal remains the same.

3 Formal Definition

In this section, I will formalize the intuitive characterization of the previous section. The purpose of this formalization is to provide a basis for the analysis of the complexity of the problem and ultimately the development of an algorithm to produce solutions to the problem.

The first task is to give a formal characterization of the features we have been discussing informally in the previous section. Many different feature systems are used by linguists and computational linguists. Two feature systems in use today are typed feature structure [Pollard and Sag, 1987; Carpenter, 1992] and the features used in DATR [Evans and Gazdar, 1990]. These systems grew out of previous work in GPSG [Gazdar *et al.*, 1985] and categories in general [Gazdar *et al.*, 1988] which in turn had roots in feature systems introduced in generative syntax [Chomsky, 1965] and generative phonology [Chomsky and Halle, 1968]. Although I will work with a specific feature system, which I will introduce shortly, the algorithms and proofs that I will present here, generalize to any feature system with the following property: whether two features clash can be computed in time polynomial in the size of the features.[3] Intuitively, two features clash if they are inconsistent. For example, the feature [fly,+] clashes with the feature [fly,-]. Different feature systems implement this notion in slightly different ways. The two feature systems just mentioned have this property. In fact, for typed feature structures, an algorithm for unification exists that is just over linear in the size of the features [Martelli and Montanari, 1982]. As we will see in a moment, unification and checking for clashes are closely related.

In the remainder of this paper, I will assume the following feature system. A feature is a pair of atomic symbols (*e.g.* [a,v]). The first element is taken from a set of attributes (ATTRIBUTES) and the second from a set of values (VALUES). For generality, I will assume that these sets are infinite. The set VALUES includes a symbol ? which intuitively specifies that the corresponding attribute is undefined or unknown for the object with the feature; more on this later. Two features clash if their attributes are the same but their values are different; this definition also holds for the value ?. Testing if two features clash can be done in constant time. I will use *clash* to denote a binary function on sets of features that produces the subset of features from the first set that are in conflict with a feature of the second (see Figure 4). I assume that the sets of features are internally consistent, *i.e.*, each set contains at most one feature per attribute. $clash(\mathcal{C}, \mathcal{D})$ can be computed for the finite sets $\mathcal{C}$ and $\mathcal{D}$ in $O(|\mathcal{C}||\mathcal{D}|)$ time by simply testing every pair formed from an element from $\mathcal{C}$ and an element of $\mathcal{D}$ for a clash. (An average time complexity of $O(|\mathcal{C}|)$ can be achieved by hashing the set $\mathcal{D}$ using attributes as keys.) If one views a typed feature structure as a set of features, then the clash function amounts to listing all the features that do not unify.

With this feature system in hand, we can move on to the problem of insertion. Remember that classes are the 'nodes' of the hierarchy, the superclasses of a class A are the classes from which A inherits, and the subclasses of A are the classes that inherit from A. An object class is a leaf of the hierarchy that represents a single object. In this section, any mention of an object class refers to the object class being constructed for the object being inserted.

As mentioned above, I am concerned with feature-based default multiple inheritance hierarchies: default in that all inheritance relationships are defeasible and multiple in that classes can inherit from more that one superclass. I assume that the hierarchies are unambiguous: for any given attribute, a node only inherits one value for it. I will discuss the issue of ambiguity with respect to insertion in section 5. I also assume, for expository reasons, that the feature sets inheritable from the classes are unique: that a given set of features cannot be inherited from more than one class in a hierarchy. The inheritance hierarchy in Figure 1 with the object node for *give* under TRANSITIVE and 3-1 is an example of a feature-based default multiple inheritance hierarchy. The OBJ-RAISING class and

[3] The features I will use in this report are all of the same constant size. However, in some feature systems, features may have complex attributes and values and thus may vary in size.

$$A = \{[a1,v1] \quad [a2,v2] \quad [a3,v3] \quad [a4,v4]\}$$

$$B = \{[a1,v5] \quad [a2,v2] \quad [a3,v20] \quad [a7,v7] \quad [a9,v12]\}$$

$$\textit{clash}(A,B) = \{[a1,v1] \quad [a3,v3]\}$$

Figure 4: An example of the *clash* function

the class for *give* both exemplify the multiple inheritance characteristic. The feature [dobj/cat,N] in the *give* class exemplifies the defeasible nature of the inheritance relationship.[4]

Further, I am concerned with cautious insertion: only the known features of the object being inserted should be inherited. An adventurous insertion would allow an object class to inherit extra features from its superclasses. I view adventurous insertion as an extension of cautious insertion.[5] In order to deal with the problem of inheriting extra features from superclasses, I will assume that the symbol ?, when used as a value in a feature, means that the value is unknown for the object or that the attribute is not appropriate for that object. I will call such features ?-features. Remember that [a,?], like any other feature, clashes with features that have the same attribute but a different value. Only object classes are allowed to contain ?-features. Figure 5 illustrates both forms of insertion and the use of the ? value. In the case of adventurous insertion, Object1 would have the feature [a3,v3] whereas with cautious insertion, its inheritance would be blocked by [a3,?].

In the previous section, I defined the insertion problem so that solutions could be incomplete: not all the features of the object being inserted need be inherited. Of course, a solution pays a price for forcing a feature to be listed in the object class. The second term of the sum in (2) specifies the price paid. Because of the possible existence of clashes (the third term), an optimal solution may be incomplete. Such incomplete insertions complicate the reduction needed to prove NP-completeness and the statement of the insertion algorithm. To circumvent this difficulty, I will assume that all hierarchies contain a singleton class for each non-?-feature of the object being inserted. (I will leave these sets out of the figures of this report.) A singleton class is a class from which exactly one feature can be inherited and which has no superclasses and no subclasses other than object classes. Listing a singleton class as a superclass in an object class is equivalent to simply listing the feature of the singleton class. This assumption allows me to require complete insertions while still being able to represent the notion of incomplete insertion since any insertion that contains a singleton set represents an incomplete insertion.

An instance of the insertion problem (IN) is a pair made up of a set $\mathcal{F}$ and a set $\mathcal{N}$ as defined below. Note that only a finite number of features is needed to specify $\mathcal{F}$ and that there is a one-to-one mapping between the sets in $\mathcal{N}$ and the classes of the hierarchy.

[4] The blocking of a default inheritance can occur between any two classes in the hierarchy, one of the classes need not be an object class.

[5] The algorithm developed in the following section would require only slight modification to produce adventurous insertions.

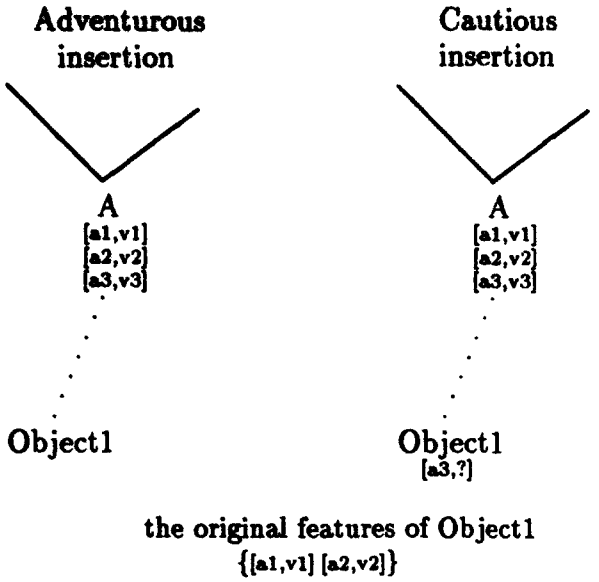


Figure 5: Adventurous vs. cautious insertion

$\mathcal{F}$ contains the features of the object being inserted. $\mathcal{F}$ must be complete in the sense that for all attributes in ATTRIBUTES there exists a feature in $\mathcal{F}$ that contains a value for that attribute. However, $\mathcal{F}$ can only have a finite number of features that contain values other than ?. All the other features are ?-features.

$\mathcal{N}$ is a set of sets of features. $\mathcal{N}$ must be finite and each of its elements must be finite. These sets cannot contain ?-features.

If we return to our discussion of the hierarchy in Figure 1, we see that the set of features of ditransitive *give*, listed in (1), is $\mathcal{F}$ for this hierarchy. The set of sets of features in (4) that resulted from compiling out the hierarchy, plus singleton sets for the features of $\mathcal{F}$, is $\mathcal{N}$.

A solution is a set $\mathcal{P} \subseteq \mathcal{N}$. $\mathcal{P}$ represents the superclasses for the object whose features are $\mathcal{F}$. Every non-?-feature in $\mathcal{F}$ must be an element of either a set in $\mathcal{P}$ or $clash(\mathcal{F}, \bigcup \mathcal{P})$.[6] The features in this latter set have to be listed explicitly for the object in order to override incorrect inheritance. An optimal solution minimizes the number of superclasses together with the number of clashes between these superclasses and $\mathcal{F}$ (see (5)[7]).

(5) $|\mathcal{P}| + |clash(\mathcal{F}, \bigcup \mathcal{P})|$

Finally, the insertion problem, **IN**, is the problem of finding optimal solutions for instances as defined above.

4 An approximation algorithm for IN

An algorithm for **IN** would take, as input, instances of **IN** and produce, as output, optimal solutions. In appendix A, I show that the decision problem for **IN**, $\mathbf{IN}_d$, is NP-complete. This result implies

[6] $\bigcup \Phi$ where Φ is a set of sets $\varphi_1, \varphi_2, ..., \varphi_n$ is equal to $\varphi_1 \cup \varphi \cup ... \cup \varphi_n$.

[7] Notice that the first two terms of (2) have been collapsed into the first term of (5). This is due to the presence of singleton sets.

that it is highly unlikely that a computationally tractable algorithm for **IN** exists. Instead, the best one should hope for is a computationally tractable algorithm that produces *near* optimal solutions.

Part of the proof in appendix A is a relatively straightforward reduction of the NP-complete set covering (**SC**) decision problem to $\mathbf{IN}_d$.[8] **SC** is defined as follows

> "...the set-covering problem consists of a finite set $\mathcal{F}$ and a family $\mathcal{N}$ of subsets of $\mathcal{F}$, such that every element of $\mathcal{F}$ belongs to at least one subset of $\mathcal{N}$...We say that a subset $\mathcal{S} \in \mathcal{N}$ covers its elements. The problem is to find a minimum-size subset $\mathcal{P} \subseteq \mathcal{N}$ whose members cover all of $\mathcal{F}$." ([Cormen *et al.*, 1990], p.974, I have substituted variable names analogous to the ones I have used for **IN**).

Figure 6 illustrates the problem and a solution. The following situation would be a real world application of **SC**: a student wishes to schedule her four years of college such that she takes the smallest number of classes but still satisfies all the requirements for a computer science degree.

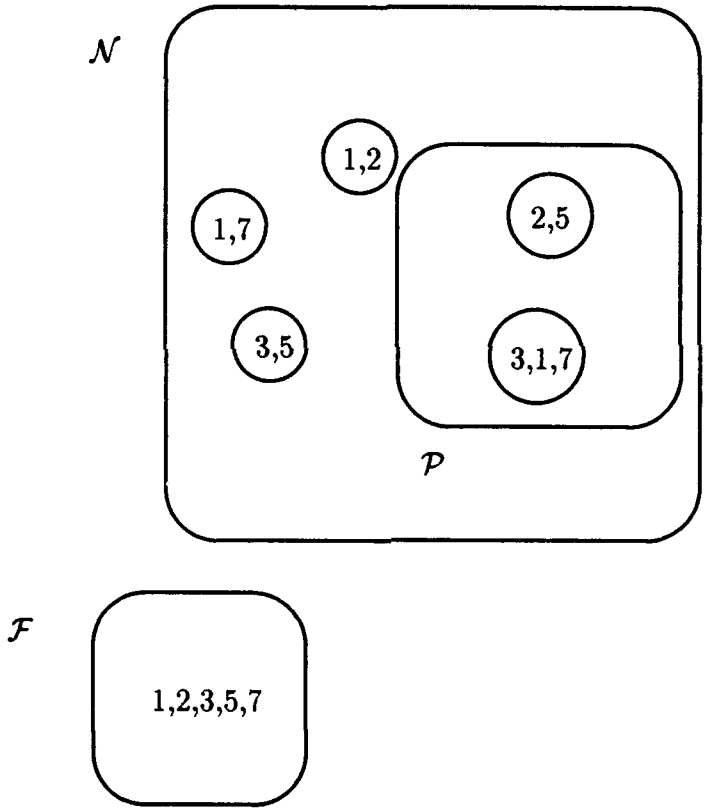


Figure 6: Set covering

IN can be viewed as extending **SC** in two ways. First, it substitutes features for integers. Features introduce the possibility of clashes between elements. This makes the process of choosing $\mathcal{P}$ more complex since one must take into account features clashed with as well as features covered. Second, intuitively, **IN** loosens the restriction that all non-?-elements in $\mathcal{F}$ must be covered. However, in the formalization of the insertion problem, I finessed this problem by requiring $\mathcal{N}$ to contain

[8] SET COVERING is also know as MINIMUM COVER. It was originally proven to be NP-complete in [Karp, 1972].

singleton sets for all non-?-elements in $\mathcal{F}$. Thus, the formalization technically requires complete coverage but retains the ability to represent incomplete covers since listing a singleton set in the object class is equivalent to listing an uncovered feature.

A polynomial time approximation algorithm exists for **SC** and the solutions it produces are guaranteed to be close to the optimal solution. More formally, the ratio of the size of the approximate solution produced by the algorithm to the optimal solution is bounded by the natural logarithm of the size of the set being covered [Lovász, 1975; Johnson, 1974; Chvátal, 1979]:

(6) $|\mathcal{P}_{approx}|/|\mathcal{P}_{optimal}| \leq ln|\mathcal{F}| + 1.$

In other words, as the size of the problem increases, the size of the approximation will grow only logarithmically faster than the size of the optimal solution.

The approximation algorithm is greedy: at any given point, it picks the subset that can cover the most features at that time and it never goes back on this choice. Greedy algorithms tend to be fast but, since they cannot backtrack, can make local choices that prevent globally optimal solutions. Because of the similarities between **SC** and **IN**, it seems likely that a greedy approximation algorithm will produce good solutions for **IN**. The algorithm `Greedy-IN` (listed below) is efficient and has produced good solutions in the small number of experiments I have performed so far (see appendix B). However, I have not yet been able to prove a ratio bound for it.

```
Greedy-IN
1.  F_temp := {[a,v] | [a,v] ∈ F ∧ v ≠ ?}
2.  P := ∅
3.  F_clash := ∅
4.  while F_temp ≠ ∅
5.      select S ∈ N that maximizes |S ∩ F_temp| - |clash(F,S) - F_clash|
6.      F_temp := F_temp - (S ∪ clash(F,S))
7.      F_clash := F_clash ∪ clash(F,S)
8.      P := P ∪ {S}
9.  return
10. list P and F_clash in the data structure for the object
```

The algorithm works as follows. It takes as input the set of features of the object being inserted, $\mathcal{F}$, and a set of sets of features, $\mathcal{N}$, that represents the classes of the hierarchies. It produces as output, a list of superclasses and a list of features that must be listed locally for the object. During each iteration of its main loop (`lines 4-9`), it picks the most suitable superclass (`line 5`). The features from $\mathcal{F}$ that can be inherited from the new superclass $\mathcal{S}$ combined with those that must be listed to block incorrect inheritance are subtracted from $\mathcal{F}$. This loop is repeated until there is no class in $\mathcal{N}$ from which more features can be inherited than must be listed to block incorrect inheritance.[9] The algorithm halts and returns the superclasses along with features that must be listed to block incorrect inheritance. To pick the most suitable superclass, a subroutine is called for each class remaining in $\mathcal{N}$ that computes the difference between the number of features remaining in $\mathcal{F}$ that can be inherited from the class and the number of new features that would be incorrectly inherited.[10]

[9] The difference being maximized in `line 5` will have a maximum value of 1 or greater because of the singleton sets in $\mathcal{N}$. The singleton sets that cover an element of $\mathcal{F}_{temp}$ will not produce any clashes and thus will have a value of 1 for the difference. When these sets are exhausted, $\mathcal{F}_{temp}$ must be empty and thus the loop will return. Remember that these singleton sets represent features that have remained uncovered and must be listed individually.

[10] Another possibility is to use the ratio of these two counts instead of the difference.

At this point, it may be helpful to step through an example insertion. Consider the hierarchy illustrated in Figure 7. It is a fragment of a hierarchy used to classify German noun compounds with respect to their stress patterns [Gibbon and Bleiching, 1991]. The compiled-out form of the hierarchy, with singleton sets, is listed in (7) and the features of the object to be inserted, the noun compound *Bürgermeister*, are listed in (8).

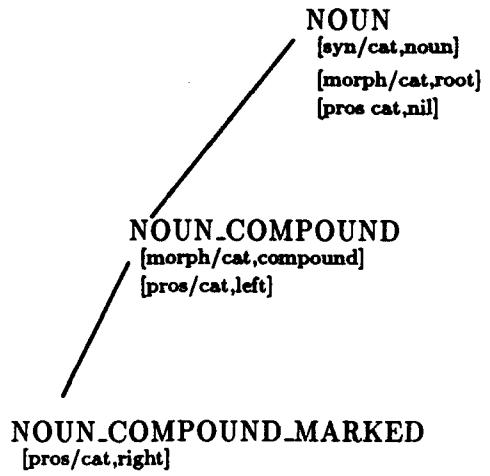


Figure 7: A hierarchy of features relevant to compound stress in German (adapted from [Gibbon and Bleiching, 1991])

(7) {{[syn/cat,noun]
[morph/cat,root]
[pros/cat,nil]}
{[syn/cat,noun]
[morph/cat,compound]
[pros/cat,left]}
{[syn/cat,noun]
[morph/cat,compound]
[pros/cat,right]}
{[determinans,Buerger]}
{[determinatum,Meister]}
{[syn/cat,noun]}
{[morph/cat,compound]}
{[pros/cat,right]}}

(8) {[determinans,Buerger]
[determinatum,Meister]
[syn/cat,noun]
[morph/cat,compound]
[pros/cat,right]}

The input to `Greedy-IN` is the set of features being inserted, $\mathcal{F}$, listed, in part[11], in (8) and the set of sets of features, $\mathcal{N}$, listed in (7). The first three steps of the algorithm initialize the variables

[11] Remember that $\mathcal{F}$ contains a potentially infinite number of ?-features. These features are not listed in (8).

to be used in the main loop: $\mathcal{F}_{temp}$ is set to the non-?-features of $\mathcal{F}$ (*i.e.* (8)), and $\mathcal{P}$ and $\mathcal{F}_{clash}$[12] to nil.

After these variables have been initialized, the main loop begins and will continue until there are no more features in $\mathcal{F}_{temp}$. In our example this will only take three iterations. During the first iteration the nodes of the hierarchy get the following scores while computing **line 5**.

NOUN: -1
NOUN_COMPOUND: 1
NOUN_COMPOUND_MARKED: 3

The singleton sets all get scores of 1. NOUN receives its score of -1 because it covers one feature, [syn/cat,noun], but clashes with two others: [morph/cat,compound] and [pros/cat,right]. Since NOUN_COMPOUND_MARKED has the highest score, **line 5** sets $\mathcal{S}$ to it.

The next set of lines (**6-8**) compute the ramifications of this new superclass. In **line 6**, [syn/cat,noun], [morph/cat,compound], and [pros/cat,right] are removed from $\mathcal{F}_{temp}$. Notice that because of this subtraction, NOUN_COMPOUND_MARKED will receive the score 0 in future iterations and therefore will not be chosen again. **Line 7** adds nothing to $\mathcal{F}_{clash}$ since the features of NOUN_COMPOUND_MARKED do not clash with any features in $\mathcal{F}$. Finally, in **line 8**, NOUN_COMPOUND_MARKED is added to $\mathcal{P}$.

Since $\mathcal{F}_{temp}$ still contains [determinans,Buerger] and [determinatum,Meister] the loop continues. In the second iteration, the following scores are produced in **line 5**.

NOUN: 0
NOUN_COMPOUND: 0

The singleton sets for the features still in $\mathcal{F}_{temp}$ receive scores of 1 while the singletons for the features already covered get scores of 0. Thus a singleton set is the winner. The choice between them is arbitrary. Let us assume the singleton set for [determinans,Buerger] is chosen: this feature is removed from $\mathcal{F}_{temp}$, $\mathcal{F}_{clash}$ remains unchanged, and the singleton is added to $\mathcal{P}$. In the next iteration the same thing occurs for the singleton set corresponding to the final feature in $\mathcal{F}_{temp}$ [determinatum,Meister]. After this iteration $\mathcal{F}_{temp}$ is empty and the loop terminates.

The final line of the algorithm produces the output for the inserted object. In this case only the first conjunct applies since no clashes occurred. The output is listed below.

```
{[syn/cat,noun]
 [morph/cat,compound]
 [pros/cat,right]}
{[determinans,Buerger]}
{[determinatum,Meister]}
```

The first set corresponds to the NOUN_COMPOUND_MARKED class and the other two correspond to singleton sets. Remember that these singleton sets represent uncovered features.

[12] $\mathcal{F}_{clash}$ is used to keep track of the features that the classes in $\mathcal{P}$ clash with. This set of features is necessary to select the new superclass since the algorithm does not want to penalize a class for clashing with a feature that has already been clashed with by a member of $\mathcal{P}$.

5 Analysis of the Algorithm

As mentioned above, **Greedy-IN** is a greedy algorithm. At choice points, greedy algorithms make a locally-optimal choice and never backtrack. As should be expected, **Greedy-IN** is fast but produces sub-optimal solutions in some situations. In this section, I will first discuss how efficiently **Greedy-IN** produces solutions. Then I will discuss the quality of the approximate solutions produced. **Greedy-IN** runs in time polynomial in the size of the encoding of the input pair $\langle \mathcal{F}, \mathcal{N} \rangle$, more specifically, $O(min\{|\mathcal{F}_{non-?}|, |\mathcal{N}|\}|\mathcal{F}_{non-?}||\mathcal{N}|)$ where $\mathcal{F}_{non-?}$ represents the set of non-? features in $\mathcal{F}$. To see this, consider the **while** loop (**lines 4-9**). It can have, at most, $min\{|\mathcal{F}_{non-?}|, |\mathcal{N}|\}$ iterations. This is because, in the worst case, either all the features in $\mathcal{F}_{non-?}$ will be covered one by one or all the classes in $\mathcal{N}$ will have been used. Each iteration has a maximum duration of $O(|\mathcal{F}_{non-?}||\mathcal{N}|)$. This is because, in order to choose the winning node $\mathcal{S}$ in **line 5**, one has to compare every feature in $\mathcal{F}_{non-?}$ to every class in $\mathcal{N}$ to see if $\mathcal{N}$ contains the feature or clashes with it. (This comparison can be done in constant time since the features in a set in $\mathcal{N}$ can be stored in an array where the attributes serve as indices into the array. An array can be used since each set in $\mathcal{N}$ is finite and only has one feature per attribute.) Thus, the time complexity of the overall algorithm is $O(min\{|\mathcal{F}_{non-?}|, |\mathcal{N}|\}|\mathcal{F}_{non-?}||\mathcal{N}|)$. The space complexity is also polynomial in the size of the input: $O(max\{|\mathcal{F}_{non-?}|, |\mathcal{N}||\mathcal{S}_{max}|\})$ where $\mathcal{S}_{max}$ is the largest element of $\mathcal{N}$. To see this notice that the algorithm must keep track of what has happened to a feature in $\mathcal{F}_{non-?}$ and it must have an array of size $|\mathcal{S}_{max}|$ to store the features of each class in $\mathcal{N}$. This space could be reduced in practice by using hash tables instead of arrays to store the features for a class in $\mathcal{N}$.

As for the relationship between the cost of a solution produced by **Greedy-IN** and the cost of a corresponding optimal solution, at this point in time, I can offer neither a theoretical proof of a logarithmic bound nor extensive experimental results showing good performance (however, a prototype implementation has performed well in small pilot studies (see appendix B)).

Instead, I will discuss the basic situation that causes sub-optimal results to be produced by greedy algorithms and two standard problems for default inheritance hierarchies from [Touretzky, 1986]: redundant links and nixon diamonds. A system that utilizes default inheritance hierarchies must address these problems since, if left unattended, they can produce inconsistencies and/or unexpected behavior in the system. I will discuss these problems in the context of insertion.

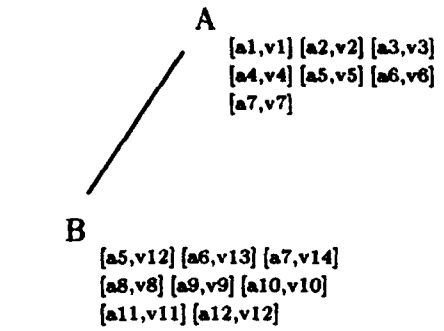
A	B	C
[a1,v1]	[a1,v1]	[a3,v3]
[a2,v2]	[a2,v2]	[a4,v4]
[a3,v3]	[a5,v5]	[a6,v6]
[a4,v4]		

$\mathcal{F} =$ {[a1,v1] [a2,v2] [a3,v3] [a4,v4] [a5,v5] [a6,v6]}

Figure 8: Problematic **IN** instance for **Greedy-IN**

An example of the basic sub-optimal solution producing situation is illustrated in Figure 8. The problem here is that class A is seductive. It covers a large number of features from $\mathcal{F}$, more than B or C. However, if A is chosen it is still necessary to pick B and C or the two appropriate singleton sets to cover the features not covered by A: [a5,v5] and [a6,v6]. If an algorithm can withstand the temptation of class A and instead go with B or C, this algorithm is rewarded by simply choosing the other class (either B or C) to cover the rest of $\mathcal{F}$. The solution set {B,C} is smaller than {A,B,C} and since both cover all of $\mathcal{F}$ and do not produce clashes, {B,C} is superior. However, **Greedy-IN** falls for sets like A every time. Thus, **Greedy-IN** would produce {A,B,C}. It might seem like one could tweak the selection criteria for the winning node (**line 5**) to produce an optimal solution and

this is, in fact, the case for any given instance. However, in general, it is always possible to come up with an instance for which these new criteria produce a sub-optimal result. This is due to the fact that the central characteristic of the algorithm stays the same: decisions are made with only local information and no backtracking is performed. Note that such problematic instances do not seem to cause **Greedy-II** to produce wildly incorrect solutions, simply slightly sub-optimal ones. In addition, it remains to be seen how often actual hierarchies contain such situations.



$\mathcal{F}$ = {[a1,v1] [a2,v2] [a3,v3] [a4,v4] [a5,v5]
[a6,v6] [a7,v7] [a8,v8] [a9,v9] [a10,v10]
[a11,v11] [a12,v12]}

Figure 9: Redundant link producing situation

Another situation that is problematic for **Greedy-II** is illustrated in Figure 9. The compiled-out set, $\mathcal{N}$, (minus the singleton sets) for this hierarchy is listed in (9) where the first set corresponds to A and second to B.

(9) $\mathcal{N}$ ={{[a1,v1] [a2,v2] [a3,v3]
[a4,v4] [a5,v5] [a6,v6]
[a7,v7]}
{[a1,v1] [a2,v2] [a3,v3]
[a4,v4] [a5,v12] [a6,v13]
[a7,v14] [a8,v8] [a9,v9]
[a10,v10] [a11,v11] [a12,v12]}}

During the first iteration of the main loop of **Greedy-II**, A will be selected to be a superclass since it will have a score of 7 whereas B will only have a score of 6. In the next round, however, B will be chosen since it will then have a score of 2 which is higher than the scores obtained by the singleton sets. At this point all the features of $\mathcal{F}$ will be covered. The new hierarchy, created by the insertion, is illustrated in Figure 10. The link from the object class to A is known as a redundant link because there is already a path to A through B. The first thing to notice is that the features that end up on the object class are the correct ones; no inconsistencies exist. This is a result of the way clashes are handled. The second thing to notice is that it is sub-optimal. The optimal insertion would list the same three features but only inherit from B. What has happened is that the choice of B canceled out the benefits of A. Thus, A became a useless superclass when B was added. In fact, the link to A could be removed without changing the features of the object class. A post-processing algorithm could be designed to search for and eliminate redundant links. However, the need for such a post-processing stage still has to be viewed as a defect of **Greedy-II**. Contrary to claim (3), that each class in a default hierarchy should be viewed as the set of features that can be inherited from this class, it seems like Figure 9 illustrates a case where the inheritance links between classes

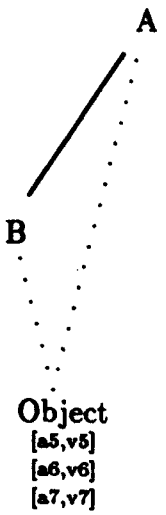


Figure 10: The redundant hierarchy

are relevant for insertion. If one had such hierarchical information one could favor lower nodes in a hierarchy over higher ones to eliminate the redundant link problem. However, it might also be possible to use weights on features in the sets of $\mathcal{N}$ to encode the relevant hierarchical structure of the classes. These weights would be added during the compilation process. For example, the weight of a feature would start at 1 in the class where it originates. It would then increase slightly each time it is passed down to a lower class. Then, in `line 5`, instead of adding 1 for each feature covered, the weight would be used. Thus, lower classes would be favored over higher ones. In any case, redundant links are a problem for **Greedy-II** as it currently stands.

The second standard problem for default inheritance hierarchies is illustrated in Figure 11. The

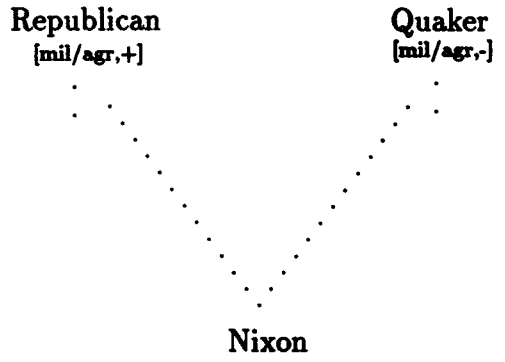


Figure 11: Nixon diamond

problem is that it is unclear what features the object class should have. More specifically, should the object class, Nixon, inherit the feature [mil/agr,+] from Republican or [mil/agr,-] from Quaker? This type of situation is know as a Nixon diamond. The question here is whether **Greedy-II** ever produces such situations. The answer is no. Because clashes are handled by listing locally the correct feature and because $\mathcal{F}$ is filled out by ?-features, it is not possible for a Nixon diamond to result from a **Greedy-II** insertion. Consider inserting the Nixon object class with an $\mathcal{F}$ such that [mil/agr,?] $\in$ $\mathcal{F}$ into a hierarchy that contains the Republican and Quaker classes as in Figure 11. If either the

Republican or Quaker class is chosen in `line 5`, then a clash will result and [mil/agr,?] will be listed in the object class for Nixon. If the other remaining class were picked as a superclass later, the local listing of [mil/agr,?] would block any possible inconsistency. If, on the other hand, the $\mathcal{F}$ for Nixon contained [mil/agr,+] and the Republican and Quaker classes were chosen by the algorithm, then at the point when Quaker was processed by **Greedy-IN**, [mil/agr,+] would be put on the clash list. Thus, it would be listed locally in the Nixon object class. An analogous sequence of events would occur if $\mathcal{F}$ contained [mil/agr,-]. Thus, Nixon diamonds are never created by **Greedy-IN**.

In this section, I started by showing that **Greedy-IN** is efficient. Next, we looked at the basic situation that causes **Greedy-IN** to produce sub-optimal insertions. Although sub-optimal, these insertions do not seem to be not wildly off the mark. Then, we looked at two standard problems for default inheritance hierarchies: redundant links and Nixon diamonds. Redundant links are produced by **Greedy-IN** and these insertions are sub-optimal. However, the insertions do not result in inconsistencies and the redundant link can be removed without affecting the features of any class. Nixon diamonds are never produced by **Greedy-IN**. Based on these results it seems that **Greedy-IN** warrants further investigation. Namely, further efforts should be made (a) to prove a logarithmic ratio bound and (b) to perform large scale experimentation with actual objects and hierarchies.

6 Conclusion

The work reported on here is motivated in part by the use of default inheritance hierarchies by many researchers to structure large natural language lexicons. An insertion system would be helpful when developing the hierarchies of a lexicon system and when trying to use such hierarchies to structure large amounts of data from outside sources. More generally, an insertion system is likely to be useful for any inheritance system. The results presented here are relevant to any system that employs default inheritance hierarchies and requires automated insertion.

I started this report by discussing informally the problem of deciding where an object belongs in a feature-based default inheritance hierarchy. Later, I formalized this insertion problem as **IN**. The crucial aspect of this formalization is that the hierarchy is viewed as a set of unrelated sets of features. In other words, each class is viewed simply as a set of features that can be inherited from it; the inheritance relations have been compiled out. Two facts combine to support this claim: i) a good insertion minimizes the space needed to store the object and thus, a class is seen as an opportunity to replace the listing of a number of features by a single superclass, ii) the structure of a default hierarchy cannot be used to reduce the search space of potential superclasses. Because **IN** is NP-complete, I designed an approximation algorithm for it: **Greedy-IN**. I showed that this algorithm is efficient and then looked at some of the possible problematic situations for the algorithm. Although more analysis and experimentation are needed, these preliminary results seem to show that the algorithm warrants such efforts. Thus, the main contributions of this work are a formalization of the insertion problem, an NP-completeness proof, and a promising greedy algorithm.

7 Acknowledgements

The ideas about insertion discussed here grew out of the need for a way to move data from the ELWIS database to inheritance hierarchies. The INSYST project prototype mentioned above is a first attempt at implementing such a system. The INSYST project was a joint effort between Sabine Reinhard, Marie Boyle-Hinrichs, and myself. Marie Boyle-Hinrichs was responsible for the implementation of the INSYST prototype. I would like to thank both of them for their efforts and their support of my own.

Both ELWIS and INSYST were developed at the University of Tübingen under the supervision of Erhard Hinrichs. I would like to thank him for the opportunity to spend a summer in Tübingen working on these projects. I would like to thank Lenhart Schubert for reading many versions of this report and providing invaluable comments. I would also like to thank Chris Barker, George Ferguson, Dafydd Gibbon, Sabine Reinhard, and Mark Young for their comments on earlier versions of this report. Finally, I would like to thank Yenjo Han, Leonidas Kontothanassis, Jeff Schneider, and Paul Dietz for their help with the initial design and analysis of `Greedy-IN`.

References

[Andry *et al.*, 1992] F. Andry, N. M. Fraser, McGlashan S., S. Thornton, and N. J. Youd, "Making DATR Work for Speech: Lexicon Compilation in SUNDIAL," *Computational Linguistics*, 18(3):245–267, 1992.

[Cahill, 1993] L. J. Cahill, "Morphonology in the Lexicon," in *Proceedings of the Sixth Conference of the European Chapter of the Association for Computational Linguistics*, pages 87–96, 1993.

[Carpenter, 1992] B. Carpenter, *The Logic of Typed Feature Structures with Applications to Unification Grammars, Logic Programs and Constraint Resolution*, Cambridge University Press, 1992.

[Chomsky, 1965] N. Chomsky, *Aspects of the Theory of Syntax*, MIT press, 1965.

[Chomsky and Halle, 1968] N. Chomsky and M. Halle, *The Sound Pattern of English*, New York: Harper and Row, 1968.

[Chvátal, 1979] V. Chvátal, "A greedy heuristic for the set-covering problem," *Mathematics of Operations Research*, 4(3):233–235, 1979.

[Copestake *et al.*, 1991] A. Copestake, V. de Paiva, and A. Sanfilippo, "The ACQUIILEX LKB: a system for repesenting lexical information extracted from machine readable dictionaries," in *Proceedings of the ACQUILEX workshop on default inheritance in the lexicon*, 1991.

[Cormen *et al.*, 1990] T. Cormen, C. Leiserson, and R. Rivest, *Introduction to Algorithms*, MIT Press, 1990.

[Evans and Gazdar, 1990] R. Evans and G. Gazdar, "The DATR Papers," Technical report, University of Sussex, Cognitive Science Research Reports, 1990.

[Feldweg and Storrer, 1992] H. Feldweg and A. Storrer, "ELWIS' lexikalische Datenbank: Beschreibung der Datenbankstruktur," Technical report, University of Tübingen, Tübingen, 1992.

[Flickinger, 1987] D. Flickinger, *Lexical Rules in the Hierarchical Lexicon*, PhD thesis, Stanford, 1987.

[Flickinger and Nerbonne, 1992] D. Flickinger and J. Nerbonne, "Inheritance and Complementation: A Case Study of *Easy* Adjectives and Related Nouns," *Computational Linguistics*, 18(3):269–309, 1992.

[Gazdar *et al.*, 1985] G. Gazdar, E. Klein, G. K. Pullum, and I. Sag, *Generalized Phrase Structure Grammar*, Harvard University Press, 1985.

[Gazdar *et al.*, 1988] G. Gazdar, G. K. Pullum, R. Carpenter, Ewan Klein, T. E. Hukari, and R. D. Levine, "Category Structures," *Computational Linguistics*, 14(1):1–19, 1988.

[Gibbon, 1990] D. Gibbon, "Prosodic association by template inheritance," in W. Daelemans and G. Gazdar, editors, *Proceedings of the Workshop on Inheritance in Natural Language Processing*, pages 65–81, 1990.

[Gibbon and Bleiching, 1991] D. Gibbon and D. Bleiching, "An ILEX model for German compound stress in DATR". FORWISS-ASL Workshop: Prosodie in der Mensch-Maschine-Kommunikation, December 1991.

[Johnson, 1974] D. S. Johnson, "Approximation algorithms for combinatorial problems," *Journal of Computer and System Sciences*, 9:256–278, 1974.

[Karp, 1972] R. M. Karp, "Reducibility among combinatorial problems," in R. E. Miller and J. W. Thatcher, editors, *Complexity of Computer Computations*. Plenum Press, 1972.

[Kilgarriff, 1993] A Kilgarriff, "Inheriting Verb Alternations," in *Proceedings of the Sixth Conference of the European Chapter of the Association for Computational Linguistics*, pages 213–221, 1993.

[Krieger and Nerbonne, 1991] H. Krieger and J. Nerbonne, "Feature-Based Inheritance Networks for Computational Lexicons," in *Proceedings of the ACQUILEX workshop on default inheritance in the lexicon*, 1991.

[Light *et al.*, 1992] M. Light, S. Reinhard, and M. Boyle-Hinrichs, "INSYST: An Automatic Inserter System for Hierarchical Lexica". unpublished manuscript, December 1992.

[Lovász, 1975] L. Lovász, "On the ratio of optimal integral and fractional covers," *Discrete Mathematics*, 13:383–390, 1975.

[Martelli and Montanari, 1982] A. Martelli and U. Montanari, "An efficient unification algorithm," *ACM Transactions on Programming Languages and Systems*, 4(2):258–282, 1982.

[Pollard and Sag, 1987] C. Pollard and I. Sag, *Information-Based Syntax and Semantics Vol. I*, University of Chicago Press, 1987.

[Reinhard, 1990] S. Reinhard, "Verarbeitungsprobleme nichtlinearer Morphologien: Umlautbeschreibung in einem hierarchischen Lexikon," in B. Rieger and B. Schaeder, editors, *Lexikon und Lexikographie*. Hildesheim: Olms, 1990.

[Russell *et al.*, 1992] G. Russell, A. Ballim, J. Carroll, and S. Warwick-Armstrong, "A Practical Approach to Multiple Default Inheritance for Unification-Based Lexicons," *Computational Linguistics*, 18(3):311–337, 1992.

[Touretzky, 1986] D. S. Touretzky, *The Mathematics of Inheritance Systems*, Morgan Kaufmann Publishers, Los Altos, CA, 1986.

8 Appendix A: Proving $\mathbf{IN}_d$ is NP-complete

An instance of the decision problem for **IN**, $\mathbf{IN}_d$, is a triple: a set $\mathcal{F}$, a set $\mathcal{N}$, and an integer **B**. $\mathcal{F}$ and $\mathcal{N}$ are as defined for **IN** in section 3. **B** represents a upper limit for the cost associated with a solution. Strictly speaking $\mathbf{IN}_d$ is a language. An instance of $\mathbf{IN}_d$ is not necessarily in $\mathbf{IN}_d$; it is merely of the right form. An instance of $\mathbf{IN}_d$ is actually in $\mathbf{IN}_d$ if there exists a solution, as defined for **IN**, that has a cost lower than **B**. The cost of a solution $\mathcal{P}$ is defined in (5) and repeated in (10).

(10) $|\mathcal{P}| + |clash(\mathcal{F}, \bigcup \mathcal{P})|$

For expository reasons, I will, from this point on, use $\mathcal{F}_{IN}$, $\mathcal{N}_{IN}$, and $\mathbf{B}_{IN}$ instead of $\mathcal{F}$, $\mathcal{N}$, and **B** when discussing $\mathbf{IN}_d$.

An instance of the decision problem for **SC**, $\mathbf{SC}_d$, is also a triple. It is made up of the following elements.

$\mathcal{F}_{SC}$ is the set of integers.

$\mathcal{N}_{SC}$ is the set of subsets of $\mathcal{F}_{SC}$ that can be used to cover $\mathcal{F}_{SC}$. For every element of $\mathcal{F}_{SC}$, there exists at least one element of $\mathcal{N}_{SC}$ that contains it.

$\mathbf{B}_{SC}$ is an integer which is the upper limit on the size of a proposed solution.

A solution for an instance of $\mathbf{SC}_d$ is a subset of $\mathcal{N}_{SC}$ that covers every element in $\mathcal{F}_{SC}$. As with $\mathbf{IN}_d$, $\mathbf{SC}_d$ is a language. An instance of $\mathbf{SC}_d$ is in $\mathbf{SC}_d$ if a solution $\mathcal{P}_{SC}$ exists that has a cost smaller than $\mathbf{B}_{SC}$. The cost for a set $\mathcal{P}_{SC}$ is its cardinality: $|\mathcal{P}_{SC}|$.

To prove that $\mathbf{IN}_d$ *is NP-complete* I have to show i) that a nondeterministic polynomial time algorithm for $\mathbf{IN}_d$ exists, ii) that there is a polynomial time (in the size of the instance) transformation from $\mathbf{SC}_d$ to $\mathbf{IN}_d$. Let us call this transformation from instances of $\mathbf{SC}_d$ to instances of $\mathbf{IN}_d$, g (see Figure 12). If an instance x of $\mathbf{SC}_d$ is in $\mathbf{SC}_d$ then $g(x)$ must be in $\mathbf{IN}_d$. If x is not in $\mathbf{SC}_d$ then $g(x)$ must not be in $\mathbf{IN}_d$.

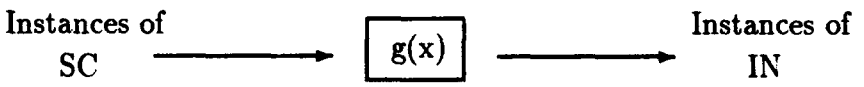


Figure 12: The transformation function g

To prove (i), consider the following nondeterministic algorithm for $\mathbf{IN}_d$: pick, at random, a set $\mathcal{P}_{IN}$ from $\mathcal{N}_{IN}$ and compare its cost to $\mathbf{B}_{IN}$, if it is smaller say "yes" otherwise say "no". This algorithm is correct because if there is a solution with a cost lower than $\mathbf{B}_{IN}$ then the computation path that corresponds to this solution will produce the output "yes". The algorithm is polynomial in the size of the input because the operation of picking a solution is clearly polynomial in the size of the input and computing the cost of this solution is also polynomial.[13] Therefore, $\mathbf{IN}_d$ is in NP.

Now on to (ii). The transformation g contains a subroutine that maps each integer in $\mathcal{F}_{SC}$ to a feature with that integer as both the attribute and value (*e.g.* 6 → [6,6]). Since none of these features will have the same attribute but different values, no clashes exist. Using this subroutine, g

[13] The second term of the cost function , $|clash(\mathcal{F}_{CL}, \bigcup \mathcal{P}_{CL})|$, may seem problematic because $\mathcal{F}_{IN}$ is infinite. However, since $\bigcup \mathcal{P}_{CL}$ is finite and only a finite number of features in $\mathcal{F}_{IN}$ are non-?-features, it is possible to compute this term in polynomial time: run through each feature in $\bigcup \mathcal{P}_{CL}$ and compare it to the non-?-features of $\mathcal{F}_{IN}$, if it clashes with a feature in $\mathcal{F}_{IN}$ then list the feature from $\mathcal{F}_{IN}$, if no non-?-feature in $\mathcal{F}_{IN}$ has the same attribute then list the corresponding ?-feature, otherwise go on to the next feature in $\bigcup \mathcal{P}_{CL}$.

performs the following mappings from sets of integers to sets of features: $\mathcal{F}_{SC}$ to $\mathcal{F}_{IN}$ and $\mathcal{N}_{SC}$ to $\mathcal{N}_{IN}$. Finally, g sets $\mathbf{B}_{IN}$ equal to $\mathbf{B}_{SC}$.

g can be computed in polynomial time with respect to the input size since the most complicated thing it does is map integers to pairs of integers. We now have to show that an x is in $\mathbf{SC}_d$ iff $g(x)$ is in $\mathbf{IN}_d$. To see this, notice that for all x that are well-formed instances of $\mathbf{SC}_d$, the sets in $g(x)$ do not contain any features that clash with any other features in these sets. This means that the cost of a solution $\mathcal{P}_{IN}$ for $g(x)$ is $|\mathcal{P}_{IN}|$ (*i.e.*, the clash term drops out). This is the same cost function as for $\mathbf{SC}_d$. In addition, for any x and $g(x)$, $\mathcal{F}_{SC}$ is isomorphic to $\mathcal{F}_{IN}$ and $\mathcal{N}_{SC}$ to $\mathcal{N}_{IN}$. Therefore, for any solution $\mathcal{P}_{SC}$ for x, a solution $\mathcal{P}_{IN}$ of the same size exists for $g(x)$. Thus, a solution $\mathcal{P}_{SC}$ exists that is smaller than $\mathbf{B}_{SC}$ iff a $\mathcal{P}_{IN}$ exists that is smaller than $\mathbf{B}_{IN}$. Therefore, x is in $\mathbf{SC}_d$ iff $g(x)$ is in $\mathbf{IN}_d$. Thus, $\mathbf{IN}_d$ is NP-complete.

9 Appendix B: Examples of INSYST in action

This appendix contains the input and output of actual runs of the INSYST program. The algorithm is similar to `Greedy-IN` but not exactly the same.

Input theory (hierarchy):

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% File:           latin.dtr
% Purpose:        Latin noun morphology (after Kennedy 1962)
% Author:         Gerald Gazdar, March 24 1989
% Documentation:  HELP *datr
% Related Files:  lib datr
%     Copyright (c) University of Sussex 1989.  All rights reserved.
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% I have modified this file slightly so that insyst can parse it.  At
% this point in time INSYST cannot perform inheritance closure for
% all the different combinations of inheritance rule types of DATR.
% (Marc Light)
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


D0:     <nom sing> == "<nva sing>"
        <nom plur> == "<nva plur>"
        <voc sing> == "<nom sing>"
        <voc plur> == "<nom plur>"
        <acc sing> == "<nva sing>"
        <acc plur> == "<nva plur>"
        <gen sing> == "<gda sing>"
        <gen plur> == "<gda plur>"
        <dat sing> == "<gda sing>"
        <dat plur> == "<gda plur>"
        <abl sing> == "<gda sing>"
        <abl plur> == "<gda plur>"
        <nva sing> == ("<root>")
        <gda plur> == ("<root>" iis)
        <nva plur> == "<gen sing>".

D1:     <> == D0
        <nva sing> == ("<root>" a)
        <acc sing> == ("<root>" am)
        <gda sing> == ("<root>" ae)
        <abl sing> == ("<root>" aa)
        <acc plur> == ("<root>" aas)
        <gen plur> == ("<root>" aarum).

D1um:   <> == D1
        <gen plur> == ("<root>" um).

D1abus: <> == D1
        <gda plur> == ("<root>" aabus).

D2:     <> == D0
        <acc sing> == ("<root>" um)
        <gda sing> == ("<root>" oo)
        <gen sing> == ("<root>" ii)
        <gen plur> == ("<root>" oorum).

D2oos:  <> == D2
```

```
        <acc plur> == ("<root>" oos).

D2us:   <> == D2oos
        <nom sing> == ("<root>" us)
        <voc sing> == D3:<abl sing>.

D2n:    <> == D2
        <nva sing> == <acc sing>
        <nva plur> == D1:<nva sing>.

D3:     <> == D0
        <gen sing> == ("<root>" is)
        <gda sing> == D2:<gen sing>
        <abl sing> == ("<root>" e)
        <nva plur> == ("<root>" ees)
        <gen plur> == D1um
        <gda plur> == ("<root>" ibus).

D3em:   <> == D3
        <acc sing> == ("<root>" em).

D3us:   <> == D3em
        <nva sing> == D2us:<nom sing>.

D3is:   <> == D3em
        <nva sing> == <gen sing>.

D3ium:  <> == D3
        <nva sing> == D3:<nva plur>
        <gen plur> == ("<root>" ium).

D3emium:<> == D3ium
        <acc sing> == D3em.

D3isium:<> == D3emium
        <nva sing> == <gen sing>.

D3im:   <> == D3isium
        <acc sing> == ("<root>" im)
        <abl sing> == D0.

D3ee:   <> == D3emium
        <abl sing> == ("<root>" ee).

D3n:    <> == D3
        <nva plur> == D2n.


D3nie:  <> == D3
        <gen plur> == D3ium
        <nva plur> == ("<root>" ia).

D3ni:   <> == D3nie
        <abl sing> == D0.

D4:     <> == D0
        <gen sing> == ("<root>" uus)
        <gda sing> == ("<root>" uu)
        <gen plur> == ("<root>" uum)
        <gda plur> == D3.

D4m:    <> == D4
        <nva sing> == D3us
        <acc sing> == D2
```

```
        <dat sing> == ("<root>" uii).

D4n:    <> == D4
        <nva sing> == <gda sing>
        <nva plur> == ("<root>" ua).

D5:     <> == D0
        <nva sing> == ("<root>" es)
        <acc sing> == ("<root>" m)
        <gda sing> == ("<root>" eii)
        <abl sing> == D3
        <nva plur> == <nva sing>
        <gen plur> == ("<root>" erum)
        <gda plur> == ("<root>" ebus).
```

Features of the item to be inserted:

```
Vis:
        <nom plur> == (vir ees)
        <nom sing> == (vis)
        <voc plur> == (vir ees)
        <voc sing> == (vis)
        <acc plur> == (vir ees)
        <dat sing> == (vir ii)
        <dat plur> == (vir ibus)
        <abl plur> == (vir ibus)
        <gen sing > == (vir is)
        <gda sing > == (vir ii)
        <nva plur > == (vir ees)
        <gda plur > == (vir ibus)
        <gen plur > == (vir ium)
        <root > == vir
        <nva sing > == vis
        <acc sing > == vim
        <abl sing > == vii.
```

The example run:

```
Command: load latin.dtr

Command: insfile latin.itml

Inserting [Vis] :

 D0 has 1 points
 D1 has -2 points
 D1um has -2 points
 D1abus has -2 points
 D2 has 0 points
 D2eos has -2 points
 D2us has -6 points
 D2n has 1 points
 D3 has 8 points
 D3em has 8 points
 D3us has 8 points
 D3is has 9 points
 D3ium has 10 points
 D3emium has 10 points
 D3isium has 11 points
 D3im has 11 points
 D3ee has 10 points
```

```
 D3n has 6 points
 D3nie has 8 points
 D3ni has 8 points
 D4 has 2 points
 D4m has 0 points
 D4n has 3 points
 D5 has 0 points
$$$$$  The winner is:

D3isium:
  uncompiled pair list for D3isium:
        <> == D3emium:<>
        <nva sing > == D3isium:<gen sing>

  compiled pair list:
        <nom sing > == vis
        <nom plur > == vir ees
        <voc sing > == vis
        <voc plur > == vir ees
        <acc plur > == vir ees
        <dat sing > == vir ii
        <dat plur > == vir ibus
        <abl plur > == vir ibus
        <gen sing > == vir is
        <gda sing > == vir ii
        <abl sing > == vir e
        <nva plur > == vir ees
        <gda plur > == vir ibus
        <gen plur > == vir ium
        <acc sing > == vir em
        <nva sing nom sing > == vis
        <nva sing nom plur > == vir ees
        <nva sing voc sing > == vis
        <nva sing voc plur > == vir ees
        <nva sing acc plur > == vir ees
        <nva sing dat sing > == vir ii
        <nva sing dat plur > == vir ibus
        <nva sing abl plur > == vir ibus
        <nva sing gen sing > == vir is
        <nva sing gda sing > == vir ii
        <nva sing abl sing > == vir e
        <nva sing nva plur > == vir ees
        <nva sing gda plur > == vir ibus
        <nva sing gen plur > == vir ium
        <nva sing acc sing > == vir em


 D0 has -3 points
 D1 has -3 points
 D1um has -3 points
 D1abus has -3 points
 D2 has -3 points
 D2oos has -3 points
 D2us has -3 points
 D2n has -2 points
 D3 has -3 points
 D3em has -3 points
 D3us has -3 points
 D3is has -2 points
 D3ium has -3 points
 D3emium has -3 points
 D3isium has 0 points
 D3im has -2 points
 D3ee has -3 points
```

```
D3n has -3 points
D3nie has -3 points
D3ni has -3 points
D4 has -3 points
D4n has -3 points
D4a has -2 points
D5 has -3 points

Final output
----------------------------------------

Vis:
        <> == D3isium:<>
        <root > == vir
        <nva sing > == vis
        <acc sing > == vim
        <abl sing > == vii

----------------------------------------
```

Input theory (hierarchy):

```
% % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % %
%
% File:            serbocro.dtr
% Purpose:         noun inflection in Serbo Croat
% Author:          Gerald Gazdar, March 24 1989
% Documentation:   HELP *datr
% Related Files:   lib datr
%      Copyright (c) University of Sussex 1989.  All rights reserved.
%
% % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % %

% not to be taken seriously as a linguistic analysis:
% it ignores relevant stress factors

% % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % %
% I have modified this file slightly so that insyst can parse it.  At
% this point in time INSYST cannot perform inheritance closure for
% all the different combinations of inheritance rule types of DATR.
% (Marc Light)
%
% % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % %

Pnoun: <nom sing> == "<root>"
<gen sing> == "<nom plur>"
<dat sing> == <nom plur>
<acc sing> == <nom sing>
<loc sing> == "<dat sing>"
<loc plur> == "<dat plur>"
<ins sing> == ("<root>" om)
<nom plur> == ("<root>" i)
<gen plur> == "<gen sing>"
<dat plur> == ("<root>" ima)
<acc plur> == "<nom plur>"
<ins plur> == "<loc plur>"
<gender> == feminine.

Fnoun: <> == Pnoun
<nom sing> == ("<root>" a)
<acc sing> == ("<root>" u)
<nom plur> == ("<root>" e)
<gen plur> == ("<root>" a)
<dat plur> == ("<root>" ama).

Mnoun: <> == Pnoun
<gen sing> == ("<root>" a)
<dat sing> == ("<root>" u)
<acc plur> == ("<root>" e)
<gender> == masculine.
```

Features of the item to be inserted:

```
Kost:
        <nom sing > == kost
        <gen sing > == (kost i)
        <dat sing > == (kost i)
        <acc sing > == kost
        <loc sing > == (kost i)
        <loc plur > == (kost ima)
        <nom plur > == (kost i)
        <gen plur > == (kost i)
        <dat plur > == (kost ima)
        <acc plur > == (kost i)
        <ins plur > == (kost ima)
        <gender > == feminine
        <root > == kost
        <ins sing > == koshcu
        <meaning > == bone.
```

The example run:

```
Command: load serbocro.dtr

Command: insfile serbocro.itm1

Inserting [Kost] :

 Pnoun has 11 points
 Fnoun has 1 points
 Mnoun has 3 points
@@@@@  The winner is:

Pnoun:
  uncompiled pair list for Pnoun:
        <nom sing > == "<root>"
        <gen sing > == "<nom plur>"
        <dat sing > == Pnoun:<nom plur>
        <acc sing > == Pnoun:<nom sing>
        <loc sing > == "<dat sing>"
        <loc plur > == "<dat plur>"
        <ins sing > == ("<root>" om)
        <nom plur > == ("<root>" i)
        <gen plur > == "<gen sing>"
        <dat plur > == ("<root>" ima)
        <acc plur > == "<nom plur>"
        <ins plur > == "<loc plur>"
        <gender > == feminine

  compiled pair list:
        <nom sing > == kost
        <gen sing > == kost i
        <dat sing > == kost i
        <acc sing > == kost
        <loc sing > == kost i
        <loc plur > == kost ima
        <ins sing > == kost om
        <nom plur > == kost i
        <gen plur > == kost i
        <dat plur > == kost ima
        <acc plur > == kost i
        <ins plur > == kost ima
        <gender > == feminine
```

```
 Pnoun has 0 points
 Fnoun has -1 points
 Mnoun has -1 points

Final output
----------------------------------------

Kost:
        <> == Pnoun:<>
        <root > == kost
        <ins sing > == koshcu
        <meaning > == bone

----------------------------------------
```